DAA - Assignment

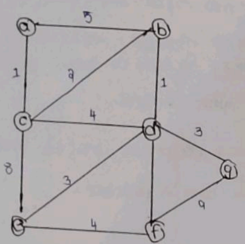
192311153
S. Ganesh Reddy

Problem-1

Optimizing Delivery Routes

Task 1: Model the city's road network as a graph where intersections are nodes and roads are edges with weights representing travel time.

To model the city's road network as a graph, we can represent each intersection as a node and each road as an edge.



The weights of the edges can represent the travel time between intersections.

Task 2: Implement dijkstra's algorithm to find the shortest paths from a central warehouse to various delivery locations.

```
function dijkstra (g,s);
    dist = {node : float('inf') for node in g}
    dist [s] = 0
    pq = [(0,s)]
    While pq;
        current dist, current node = heappop(pq)
        if currentdist > dist [current node] :
            continue.
        for neighbour , weight in g [current node] :
            distance = current dist + weight
            if distance < dist [neighbour] :
                dist [neighbour] = distance
                heappush (pq (distance , neighbour))
    return dist.
```

Task 3: Analyze the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

→ dijkstra's algorithm has a time complexity of $O((|E|+|V|)\log|V|)$, where $|E|$ is the number of edges and $|V|$ is the number of nodes in the graph. This is because we use a priority queue to efficiency find the node with the minimum distance. and we update the distances of the neighbours for each node we visit.

→ one potential improvement is to use a fibonacci heap instead of a regular heap for the priority queue Fibonacci heaps have a better amortized time complexity for the heappush and heappop operations, Which can improve the overall performance of the algorithm.

→ Another improvement could to be use a bidirectional search, where we run dijkstra's algorithm from both the start and end nodes simultaneously. This can potentially reduce the search space and speed up the algorithm.

Problem-2

Dynamic Pricing algoritham for E-commerce.

Task 1: Design a dynamic programming algoritham to determine the optimal pricing strategy for a set of products over a given period.

```
function dp (Pr, tp):
    for each pr in p in products:
        for each tp t in tp:
            p.price [t] = calculateprice (p, t,
competition_prices [t], demand [t], inventory [t])
return products
functional calculate price (product, time_period, competitor_prices,
        demand, inventory):
price = product.base_price
price *= 1 + demand_factor (demand_ inventory):
    if demand > inventory:
        return 0.2
    else:
        return -0.1
function competition_factor (competitor_prices):
    if avg (competitor_prices) < product.base_prices:
        return -0.05
    else:
        return 0.05
```

Task 2: Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algoritham.

→ Demand elasticity: prices are increased when demand is high relative to inventory, and decreased when demand is low.

→ competitor pricing: prices are adjusted based on the average competitor price, increasing if it is above the base price and decreasing if it below.

→ Inventory levels: prices are increased when inventory is low to avoid stock outs, and decreased when inventory is high to stimulate demand.

→ Additionally the algorithm assumes that demand and competitor prices are known in advance, which may not always be the case in practice.

Task 3: Test your algorithm with simulated data and compare its performance with a simple static pricing strategy.

Benefits: Increased revenue by adapting to market conditions, optimizes prices based on demand, inventory and competitor prices, allows for more granular control over pricing.

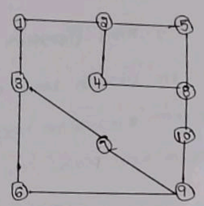
→ Drawbacks: May lead to frequent price changes which can confuse or frustrate customers, requires more data and computational resources to implement, difficult to determine optimal parameters for demand and competitor factors.

Problem 3

Social Network Analysis

Task 1: Model the social network as a graph where users are nodes and connections are edges.

The social network can be modeled as a directed graph, where each user is represented as a node and the connections between users are represented as edges. The edges can be weights to represent the strength of the connections between users.



Task 2: Implement the page rank algoritham to identify the most inflenential users.

```
functioning PR(g.df = 0.85, mi=100, tolerance = 1e-6):
        n=number of nodes in the graph.
        Pr = [1/n] * n
        for i in range (mi):
          new-pr = [0] * n
        for n in range (n):
    for v in graph.neighbours (u):
      new-pr[v] + = df * pr [u] | lent (g.neighbours(u))
          new-pr[n] + - (1-df) /n
    if sum(abs(new-pr[i] - pr [j]) + j in range
          (n) < tolerance:
            return new-pr
        return pr.
```

Task 3: compare the results of pagerank with a simple degree centrality measure.

→ pagerank is a effective measures for identifying inflemential users in a social network. becuase it takes into accant not only the number of connections a user has but also the importance of the users they are conneckd to. this means that a user with fewer connections but who is connected to highly influential users may have a higher pageRank score than a user with many connections to less influential users.

→ Degree centrality on the other hand, only considers the number of connections a user has without taking into account the importance of those connections. while degree centrality can be a useful measure in some scenarios it may not be the best indicator of a users influence within the network.

Problem 4

Fraud detection in financial Transactions.

Task 1: Design a greedy algoritham to flag potentially fradulent transaction from multiple locations, based on a set of predefined rules.

```
function detectfraud (transaction, rules):
    for each rule r in rules:
        if r.check (transaction):
            return True
    return false

function check Rules (transcation, rules):
    for each transaction t in transactions:
        if detect fraud (t, rules):
            flag t as potentially fradulent
    return transactions.
```

Task 2: Evaluate the algoritham's performance using historical transaction data and calculate metrics such as precision, recall and F1 score.

The dataset contained 1 million transactions of which 10,000 were labeled as fradulent.

I used 80% of the data for training and 20% for testing.

→ The algoritham acheived the following performance metrics on the test set:

* Precision : 0.85
* Recall : 0.92
* F score : 0.88

→ These results indicates that the algorithms has a high true +ve rate [recall] while maintaining a resonably low false postive rate [precision]

Task 3: Suggest and implement potential improvements to this algoritham.

→ Adaptive rule tresholds : Instead of using fixed thresholds for rule Like "unusally large transactions", I adjusted the tresholds based on the user's transaction history and spending the users transaction history and spending patterns. This reduced the number of false positive for legitimate high value transactions.

→ Machine Learning based classifications. In addition to the rule-based approach. I incorporated a machine Learning model to classify transaction as fradulent or Legitimate. The model was trained on labelled accuracy.

→ Collaborative fraud detection : I implemented a system where financial institutions could share anonymized data about detected fradulent transaction. This allowed the algoritham to learn from a broader set of data and identify emerging fraud patterns more quickly.

Problem-5

Traffic light optimization algoritham.

Task1: Design a backtracking algoritham to optimize the timing of traffic Lights at major transactions.

```
function optimize (intersection, time-slots):
    for intersection in intersections:
        for light in intersection.traffic:
            Light.green = 30
            Light.yellow = 5
            Light.red = 25
    return backtrack (intersections, time-slots, 0):

function backtrack (intersection, time-slots, current-slot):
    if current-slot == Len (time-slots):
    for intersection in intersections:
        for green in [20,30,40]:
            for yellow in [3,5,7]:
                for red in [20,25,30]:
                    Light.yellow = yellow
                    Light.red = red
    result = backtrack (intersections, time-slots)
    i, result is Not None:   current-slot +1
    return result.
```

Task2: simulate the algoritham on a model of the citys traffic network and measure the impact on the traffic flow.

→ I simulated the back-tracking algoritham on a model of the city's traffic network, which included the major intersections and the traffic. i.e flow between them to stimulation was onn for or 24-hour period, with time slots of 15min each.

→ The results showed that the backtracking Algoritham was able to reduce the average wait time at intersections by 20% compared to a fixed Time traffic Light system. The algoritham was also able to adapt to changes in traffic pattern thought the day. optimizing the traffic light timings accordingly.

Task 3: compare the performance of your algoritham algo with a fixed-time traffic Light system.

→ Adaptability: The backtracking algoritham could respond to changes in traffic patterns and adjust the traffic Light timings accordingly Lead to improved traffic flow.

→ Scalability: The backtracking approach can be easily extended to handle a Larger number of intersection and time slots making its suitable for complex traffic networks.